Verified Compilation of Linearizable Data Structures*

Mechanizing Rely Guarantee for Semantic Refinement

Yannick Zakowski David Cachera Delphine Demange David Pichardie
Univ Rennes / Inria / CNRS / IRISA

ABSTRACT

Compiling concurrent and managed languages involves implementing sophisticated interactions between client code and the runtime system. An emblematic runtime service, whose implementation is particularly error-prone, is *concurrent garbage collection*. In a recent work [31], we implement an on-the-fly concurrent garbage collector, and formally prove its functional correctness in the Coq proof assistant. The garbage collector is implemented in a compiler intermediate representation featuring abstract concurrent data structures.

The present paper extends this work by considering the concrete implementation of some of these abstract concurrent data structures. We formalize, in the Coq proof assistant, a theorem establishing the semantic correctness of a compiling pass which translates abstract, atomic data structures into their concrete, fine-grained concurrent implementations.

At the crux of the proof lies a generic result establishing once and for all a simulation relation, starting from a carefully crafted rely-guarantee specification. Inspired by the work of Vafeiadis [28], implementations are annotated with linearization points. Semantically, this instrumentation reflects the behavior of abstract data structures.

1. INTRODUCTION

Verified compilation is slowly becoming a reality. CompCert [21] is a realistic and fully verified C compiler. It is now commercially available and makes possible the generation of optimized code for highly critical applications, whereas in the past compiler optimizations were often completely switched off in this context [20]. The situation for managed languages, that require a runtime environment including e.g. automatic memory management, is quite different. Indeed, there is still a long way to go before a realistic and fully verified compiler becomes available, especially in a concurrent setting.

One major challenge in this landscape is the verification of an executable runtime system for such languages and an emblematic runtime service is garbage collection. In [31], we recently presented a Coq formalization of a fully concurrent garbage collector, where mutators never have to wait for the collector. This work and other similar proof efforts [25, 10, 9, 7, 8, 11, 12] are important stepping stones towards a realistic, verified garbage collector but none of these projects is ready yet to provide an *executable* verified artifact.

Our approach in [31] is based on a dedicated intermediate representation (IR) that features strong type guarantees, dedicated support for abstract concurrent data structures, and high-level iterators on runtime internals (e.g. objects, thread ids). We have implemented a realistic implementation of Domani et al.'s GC algorithm [3] in this IR and use its companion Rely-Guarantee logic to prove its functional correctness. But in order to make this runtime service executable, we need to provide a verified compiler for our IR. The present paper is in line with this objective: we provide a mechanized theorem to prove the soundness of compilation passes implementing abstract concurrent data structures. Using this result, the correctness proof of runtime services can then be propagated down to their low-level implementation.

To fit into a classical verified compiler infrastructure [21], we need a correctness theorem for a compiler from a source language with atomic data-structures, to a target language where data-structures are implemented with fine-grained concurrency. The standard theorem in the community proves that the compiler preserves the behavior of source programs: for any source program p,

$$\texttt{obs}(\texttt{compile}(p)) \subseteq \texttt{obs}(p) \tag{1}$$

where `obs` denotes the observable behaviors of a program.

Fine-grained concurrency allows programmers to reduce synchronization costs to a minimum but renders the programming activity extremely subtle and error-prone, due to race conditions. Experts who program fine-grained concurrent algorithms generally resort to well-chosen data structures that can be accessed using *linearizable* methods: even though the implementation of these methods is by no means atomic, they appear to the rest of the system to occur instantaneously. Linearizability thus considerably helps abstraction: the programmer can safely reason at a higher-level, and assume an abstract, atomic specification for these data structures.

Initially, linearizability was formally defined by Herlihy and Wing [16]. In their seminal paper, they model system executions as sequences, or histories, of operation invocation and response events, and a system is linearizable whenever all valid histories can be reordered into sequential histories. In this work, we rather consider an alternative formulation, based on semantic refinement that is well aligned with the standard correctness criterion in verified compilation [21].

*This work was supported by Agence Nationale de la Recherche, grant number ANR-14-CE28-0004 DISCOVER.
SAC 2018,April 09-13, 2018, Pau, France

Proving a statement like (1) is done by showing that the target program $\texttt{compile}(p)$ simulates the source program p: for any execution of the target program, we must exhibit a matching execution of the source program. While the definition of the matching relation is relatively intuitive, dealing with the formal details can be cumbersome, and further, proving that it is indeed maintained along the execution is difficult. Hence, we would like to resort to popular program verification techniques to lighten the proof process.

In [28], Vafeiadis proposed a promising approach that fits our needs. It is based on Rely-Guarantee (RG) [18], a popular proof technique extending Hoare logic to concurrency. In RG, threads interferences are described by binary relations on shared states. Each thread is proved correct under the assumption that other threads obey a *rely* relation. The effect of the thread must respect a *guarantee* relation, which must be accounted for in the relies of the other threads. RG allows for thread-modular reasoning, and hence removes the need to explicitly consider all interleavings in a global fashion. Now, RG alone does not capture the notion of a linearizable method. So Vafeiadis proposes to extend RG to *hybrid* implementations, i.e. fine-grained concurrent methods instrumented with linearization points that reflect the abstract, atomic behavior of methods in a ghost part of the execution state. The approach is elegant, but is not formally linked to the standard notion of compiler correctness. Recently, Liang et al. [22] improved on the work of Vafeiadis by expressing the soundness of the methodology with a semantic refinement. Their work undoubtedly makes progress in the right direction. Unfortunately, their proof is not machine-checked.

To sum up, we provide a machine-checked semantic foundation to the approach outlined in Vafeiadis's PhD, and embed it in a generic, end-to-end compiler correctness theorem. More precisely, we make the following contributions:

- We integrate the notion of linearizable data structures in a formally verified compiler. Correctness is phrased in terms of a simple semantic refinement and avoids the difficult, though traditional, definition of linearizability.
- Our proof is generic in the abstract data structures under consideration, and in the source program using them. The underlying simulation is proved once and for all, provided that hybrid implementations meet a certain specification.
- We express this specification in terms of RG reasoning, so it integrates smoothly with deductive proof systems.

All theorems are proved in Coq, and available online [30].

2. CHALLENGES AND OVERVIEW

At the source level, we use a core concurrent language $\mathcal{L}^\sharp$ that features *abstract data structures* with a set $\mathcal{I}$ of atomic methods. We program a compiler $\texttt{compile}\langle\mathcal{I}\rangle \in \mathcal{L}^\sharp \to \mathcal{L}$, which replaces the abstract data structures with their fine-grained concurrent implementation in $\mathcal{L}$. Our goal is to prove that this compiler is correct, in the sense that it preserves the observable behaviors of source programs, with a theorem of the form

$$\forall p \in \mathcal{L}^\sharp,\ \texttt{obs}(\texttt{compile}\langle\mathcal{I}\rangle(p)) \subseteq \texttt{obs}(p) \qquad (2)$$

Our main result is generic in the abstract data structures used in $\mathcal{L}^\sharp$ and their implementation $\mathcal{I}$. Here, we illustrate on a simple pedagogical example what are the intrinsic challenges, and briefly overview our technical contribution. A more challenging example is described in Section 6.

2.1 Simple Lock Example

Suppose $\mathcal{L}^\sharp$ offers abstract locks, each providing two methods $\mathcal{I} = \{\texttt{acquire}, \texttt{release}\}$. At the level of $\mathcal{L}^\sharp$, an abstract lock can be seen as a simple boolean value, with the expected atomic semantics. At the concrete level, $\mathcal{L}$, `acquire` and `release` are implemented with the code in Figure 1. They use a boolean field `flag`, denoting the status of the lock. Releasing the lock simply sets the field to 0, while acquiring it uses a *compare-and-swap* instruction (`cas`) to ensure that two threads do not simultaneously acquire a shared lock. Note that both methods could be executed concurrently by two client threads.

```
def acquire() ::=
  ok = 0;
  do {
    ok=cas(this.flag,0,1)
  } while (ok == 0);
  return

def release() ::=
  this.flag = 0;
  return
```

Figure 1: Spinlock in $\mathcal{L}$.

2.2 Semantic Refinement

Proving a theorem like (2) is done with a simulation: for any execution of the target program, we must exhibit a matching execution of the source program. Between $\mathcal{L}$ and $\mathcal{L}^\sharp$, the simulation is however particularly difficult to establish.

We illustrate the situation with Figure 2. Execution steps

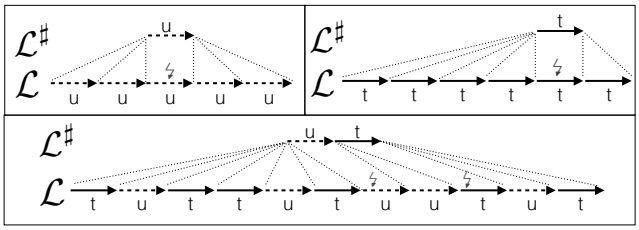

Figure 2: Intra and inter-thread matching step relations.

labeled with $\lightning$ are those where the effect of a method in $\mathcal{I}$ becomes visible to other threads, and thus determine the behavior of other methods in $\mathcal{I}$ executed concurrently.

At the intra-thread level (Figure 2, top), we need to relate several steps of a thread in the target program to a single step in the source. The situation is even more difficult at the inter-thread level (Figure 2, bottom): the interleaving of threads at the target level (first `u`, then `t` in the example)

must sometimes be matched at the source level by another interleaving (first `t` then `u` in the example). Indeed, it all depends on which thread will be the first to execute its $\lightning$ step in the concrete execution. The matching step for a given thread hence depends on the execution of its environment.

Our main result, that we present in Section 4, removes this difficulty by establishing, under some hypotheses, a *generic* simulation that entails semantic refinement. This is a meta-theorem that we establish once and for all, independently of the abstract data-structures available in $\mathcal{L}^\sharp$.

2.3 Rely-Guarantee for Atomicity

To prove the atomicity of linearizable methods, and establish the simulation, we introduce an intermediate, proof-dedicated language $\mathcal{L}^\flat$, that comes with an RG proof system.

$\mathcal{L}^\flat$ provides *explicit linearization point* annotations, $\texttt{Lin}(b)$, to guide the proof. In the spirit of Vafeiadis's methodology, $\texttt{Lin}(b)$ annotations have an operational effect: they trigger, whenever condition b is met, the execution of the abstract method in a ghost part of the state (they are equivalent to skip if the condition is not met). `Lin` instructions are what makes $\mathcal{L}^\flat$ *hybrid*. Additionally, they allow to track whether a thread has reached the linearization point or not, thus helping the construction of our generic simulation. The annotated spinlock code is shown in Figure 3.

```
def acquire() ::=
 ok = 0;
 do {
   atomic⟨ok=cas(this.flag,0,1);Lin(ok==1)⟩
 } while (ok == 0);
 return

def release() ::=
  atomic⟨this.flag = 0; Lin(true)⟩;
  return
```

Figure 3: Spinlock in $\mathcal{L}^\flat$.

Considering the intermediate language $\mathcal{L}^\flat$, our compiler is in fact defined as $\texttt{compile} = \texttt{clean} \circ \texttt{concretize}$. It first transforms a source program p into $\texttt{concretize}\langle\mathcal{I}\rangle(p) \in \mathcal{L}^\flat$ where abstract methods are implemented by hybrid, annotated code, supplied by the programmer of the data-structure. Then, a second compilation phase $\texttt{clean}\langle\mathcal{I}\rangle \in \mathcal{L}^\flat \to \mathcal{L}$ takes care of removing `Lin` instructions in the target program.

We prove in Coq that the compiler is correct, provided that hybrid methods in $\mathcal{I}$ are proved correct *w.r.t.* an RG specification, RGspec, that we carefully define in terms of $\mathcal{L}^\flat$ semantics to prove:

$$\mathsf{RGspec}\,(\mathcal{I}) \Longrightarrow \forall p \in \mathcal{L}^\sharp,\ \texttt{obs}(\texttt{compile}\langle\mathcal{I}\rangle(p)) \subseteq \texttt{obs}(p)$$

via the aforementioned simulation. We also embed in judgment RGspec the requirements sufficient to show that `Lin` instructions can be safely removed by the `clean` phase. This ensures that, despite their operational nature, `Lin` instructions are only passively instrumenting the program and its semantics.

2.4 Using our Result

The typical workflow for using our generic result is to (i) define the abstract data structures specification, i.e. their type, and the atomic semantics of methods in $\mathcal{I}$, (ii) provide a concrete implementation, i.e. their representation in the heap and a fine-grained hybrid implementation of methods, (iii) define a coherence invariant between abstract and concrete data structures that formalizes the link between a concrete data structure and its abstract view, (iv) define the rely and guarantee of each method, (v) prove the RG specification of each method using a dedicated program logic[1] and (vi) apply our meta-theorem to get the global correctness result.

We have successfully used this workflow to prove the correctness of the above spinlock example, as well as concurrent buffers, a data structure we used [31] in our implementation of a verified concurrent garbage collector.

3. LANGUAGE SYNTAX AND SEMANTICS

As explained in the previous section, this work considers three different languages $\mathcal{L}^\sharp$, $\mathcal{L}^\flat$ and $\mathcal{L}$. To lighten the presentation, however, we will just assume one language, $\mathcal{L}^\flat$, that includes all features, and keep in mind that source programs in $\mathcal{L}^\sharp$ do not include any linearization instrumentation, while target programs in $\mathcal{L}$ do not contain abstract method calls or linearization instrumentation.

$\mathcal{L}^\flat$ is a concurrent imperative language, with no dynamic creation of threads. It is dynamically typed, and features a simplified object model: objects in the heap are just records, and rather than virtual method calls, the current object – the object whose method is being called – is an extra function argument, passed in the reserved variable `this`. In the sequel, *Var* is a set of variable identifiers, method names range over $\texttt{m} \in \mathit{Methods}$, and field identifiers range over $\texttt{f} \in \mathit{Fields}$.

3.1 Values and Abstract Data Structures

We use the domain of values $\mathit{Val} = \mathbb{Z} + \mathit{Ref} + \mathit{Null}$, where *Ref* is a countable set of references. A central notion in the language is that of abstract data structure. They are specified with an atomic specification. All our development and our proofs are parameterized by an abstract data structure specification. It could be abstract locks as in Section 2, bags, stacks, or buffers as detailed in Section 6.

Definition 3.1. *An* abstract data structure *is specified by a tuple $(A^\sharp, \mathcal{I}, [\![.]\!]^\sharp, \mathcal{P})$ where $A^\sharp$ is a set of abstract objects; $\mathcal{I} \subseteq \mathit{Methods}$ is a set of abstract methods identifiers, whose atomic semantics is given by the partial map $[\![.]\!]^\sharp \in \mathcal{I} \to (A^\sharp \times \mathit{Val}) \hookrightarrow (A^\sharp \times \mathit{Val})$, taking as inputs an object and a value, and returning an updated object and a value; and $\mathcal{P} \subseteq \mathit{Fields}$ reserves private field identifiers for the concrete implementation of abstract methods in $\mathcal{I}$.*

Abstract objects in $A^\sharp$ are the possible values that an instance of a data structure can take. We use private fields to express the property of *interference freedom* from Herlihy and Shavit [15]. Namely, client code can only use public

[1]The program logic is a contribution that is out of the scope of this paper, though we use it in our development.

```
<expr>    e ::= n | null | x | e + e | - e | e mod n | ...
<bexpr>   b ::= true | e == e | e <> e | b || b | !b | ...
<comm>    c ::= • | assume(b) | print(e) | x = e
                | x = y.f | x.f = y | x = new(f,...,f)
                | return(e) | x = y.m(z)
                | c ; c | c + c | loop(c) | atomic ⟨c⟩
<comm>♯   c♯ ::= c | x =# y.m(z)
<comm>♭   c♭ ::= c | Lin (b)
```

Figure 4: Language Syntax.

fields in $Fields \setminus \mathcal{P}$, and concrete implementations of abstract methods in $\mathcal{I}$ use private fields only.

EXAMPLE 3.1. *In the spinlock example, the lock abstract data structure specification is given by a set of abstract objects* $A^\sharp = \{\texttt{Locked}, \texttt{Unlocked}\}$*, and an abstract methods set* $\mathcal{I} = \{\texttt{acquire}, \texttt{release}\}$*. Lock implementations use a single private field* $\mathcal{P} = \{\texttt{flag}\}$*. The atomic, abstract semantics of lock methods are defined as*

$$[\![\texttt{acquire}]\!]^\sharp(\texttt{Unlocked}, v) \;=\; (\texttt{Locked}, \mathit{Null})$$

for any value v *(only an* `Unlocked` *lock can be acquired), and*

$$[\![\texttt{release}]\!]^\sharp(l, v) \;=\; (\texttt{Unlocked}, \mathit{Null})$$

for any input abstract lock l *and value* v*.*

3.2 Language Syntax

The syntax of the language is detailed on Figure 4. In the sequel, we fix an arbitrary abstract data structure specification $(A^\sharp, \mathcal{I}, [\![.]\!]^\sharp, \mathcal{P})$. The language provides constants ($\texttt{n}, \texttt{null}, \texttt{true}\ldots$), local variables ($\texttt{x}, \texttt{y}, \texttt{z}\ldots$), and arithmetic and boolean expressions ($\texttt{e}, \texttt{b}$). Regular commands ($\texttt{c}$) are standard, and common to the three languages. They include • (skip), an `assume`(e) statement, a `print`(e) instruction that emits the observable value of `e`, variable assignment of an expression, field reads and updates, record allocation, nondeterministic choice (+), loops, and atomic blocks `atomic` $\langle c \rangle$. Concrete method calls are written `x = y.m(z)`.

Some instructions are specific to a language level. In the source language $\mathcal{L}^\sharp$, abstract method calls on a abstract object are written `x =# y.m(z)`. For any `m` $\in \mathcal{I}$, such a call in a $\mathcal{L}^\sharp$ program is compiled to a concrete call `x = y.m(z)` in the $\mathcal{L}$ program. In $\mathcal{L}^\flat$, the `Lin`(b) instruction is used to annotate a linearization point.

Finally, a client program is defined by a map from method names in $Methods \setminus \mathcal{I}$ to their command, and a map from thread identifiers to their command. In the sequel, we will write `m`.*comm* for getting the command of method `m`, leaving the underlying program implicit.

3.3 Language Semantics

For the sake of conciseness, we present here a partial view of the semantics, and refer the reader to the formal development [30] for full details[2].

[2]In our formal development, we use a continuation-based semantics to handle atomic blocks and method calls. This has proven to lighten the mechanization of many proofs, by removing any recursivity from the small step semantics.

We assume a standard semantics $[\![\cdot]\!]$ for expressions, omitted here. Abstract objects are stored in an abstract heap, ranged over by $h^\sharp \in H^\sharp = \mathit{Ref} \to A^\sharp$. At the concrete level, abstract objects are implemented by regular, concrete objects, living in a concrete heap $h \in H = (\mathit{Ref} \times \mathit{Fields}) \to \mathit{Val}$. A shared memory, ranged over by $\sigma \in H^\sharp \times H$ is made of an abstract heap and a concrete heap.

An intra-thread state $ts = \langle \texttt{m}, \texttt{c}, l, ls \rangle$ includes a current method `m`, a current command `c`, a local environment $l \in \mathit{Lenv} = \mathit{Var} \to \mathit{Val}$, and a linearization state $ls \in \mathit{LinState}$, that we explain below. The intra-thread operational semantics, partially shown in the top four rules of Figure 5, is a transition relation $\cdot \xrightarrow{\cdot} \cdot$ on intra-thread states. It is labeled with observable events ranged over by o. An observable event is either a numeric value or the silent event τ.

The print instruction (rule PRINT) is the only one that emits an observable value, namely the value of the expression that is printed. Neither the local state nor the shared memory are modified by this instruction. Print instructions are only allowed outside abstract methods implementations.

An abstract method call (rule ACALL) `x =# y.m(z)` is executed according to the abstract semantics $[\![\texttt{m}]\!]^\sharp$, and modifies only the abstract heap.

Concrete method calls (rule CCALL) behave as expected, but additionally manage the local linearization state. This linearization state notably keeps track of whether the execution of the current method is before its linearization point (Before) or not (After). Initially, the linearization state is set to Nolin. When control transfers to a method in $\mathcal{I}$ through a concrete method call, the linearization state changes from Nolin to Before (see rule CCALL). It switches to After when executing a `Lin` instruction (rule LINTRUE) when the linearization condition `b` is true, and then back to Nolin on method return. Linearization states are used in the simulation proof, and instrument $\mathcal{L}^\flat$ only.

At the $\mathcal{L}^\flat$ level, the `Lin` instruction also accounts for the effect on the abstract heap of concrete methods in $\mathcal{I}$: it performs the abstract atomic call $[\![\texttt{m}]\!]^\sharp$ to the enclosing method `m`, updating the local environment and abstract heap.

The interleaving of threads is handled in rule INTL, with relation $(\gamma, \sigma) \xrightarrow{o} (\gamma', \sigma')$ between global states (γ, σ), where γ maps thread identifiers to thread local states and σ is a shared memory. Mutual exclusion between atomic blocks is ensured by the $\neg \mathit{inAtomic}$ side condition.

Finally, program behaviors are defined on top of the interleaving semantics, as expressed by the following definition.

DEFINITION 3.2. *The* observable behavior *of a program* p*, written* `obs`(p, σ_i)*, is either a finite trace of values emitted by a finite sequence of transitions or a infinite trace of values emitted by an infinite sequence of transitions, from an initial shared memory* σ_i*.*

4. MAIN THEOREM

In this section, we formalize our main result. We use the following notations and vocabulary. For a set A, an A predicate P is a subset of A. An element $a \in A$ satisfies the A predicate

$$\textsc{Print}\ \frac{[\![\mathtt{e}]\!]l = v \quad \mathtt{m} \notin \mathcal{I}}{(\langle \mathtt{m}, \mathtt{print(e)}, l, ls\rangle, \sigma) \xrightarrow{v} (\langle \mathtt{m}, \bullet, l, ls\rangle, \sigma)}$$

$$\textsc{ACall}\ \frac{l(\mathtt{y}) = r \quad h^\sharp(r) = a \quad l(\mathtt{z}) = v \quad \mathtt{m}' \in \mathcal{I} \quad [\![\mathtt{m}']\!]^\sharp(a, v) = (a', v')}{(\langle \mathtt{m}, \mathtt{x=\#\,y.m'(z)}, l, ls\rangle, (h^\sharp, h)) \xrightarrow{\tau} (\langle \mathtt{m}, \bullet, l[\mathtt{x} \mapsto v'], ls\rangle, (h^\sharp[r \mapsto a'], h))}$$

$$\textsc{CCall}\ \frac{\begin{array}{c} l(\mathtt{y}) = r \quad l(\mathtt{z}) = v \quad l' = [\mathtt{m}'.\mathtt{this} \mapsto r,\ \mathtt{m}'.\mathtt{arg} \mapsto v] \\ ls' = \text{ if } \mathtt{m}' \in \mathcal{I} \text{ then } \mathsf{Before}(r, v) \text{ else } \mathsf{Nolin} \end{array}}{(\langle \mathtt{m}, \mathtt{x} = \mathtt{y.m'(z)}, l, \mathsf{Nolin}\rangle, \sigma) \xrightarrow{\tau} (\langle \mathtt{m}', \mathtt{m}'.comm, l', ls'\rangle, \sigma)}$$

$$\textsc{LinTrue}\ \frac{[\![\mathtt{b}]\!]l = true \quad h^\sharp(r) = a \quad \mathtt{m} \in \mathcal{I} \quad [\![\mathtt{m}]\!]^\sharp(a, v) = (a', v')}{(\langle \mathtt{m}, \mathtt{Lin(b)}, l, \mathsf{Before}(r, v)\rangle, (h^\sharp, h)) \xrightarrow{\tau} (\langle \mathtt{m}, \bullet, l[\mathtt{x} \mapsto v'], \mathsf{After}(r, v, v')\rangle, (h^\sharp[r \mapsto a'], h))}$$

$$\textsc{LinFalse}\ \frac{[\![\mathtt{b}]\!]l = false}{(\langle \mathtt{m}, \mathtt{Lin(b)}, l, \mathsf{Before}(r, v)\rangle, (h^\sharp, h)) \xrightarrow{\tau} (\langle \mathtt{m}, \bullet, l, \mathsf{Before}(r, v)\rangle, (h^\sharp, h))}$$

$$\textsc{Intl}\ \frac{\gamma(t) = ts \quad (ts, \sigma) \xrightarrow{o} (ts', \sigma') \quad \forall t' \neq t,\ \neg inAtomic(\gamma(t'))}{(\gamma, \sigma) \xrightarrow{o} (\gamma[t \mapsto ts'], \sigma')}$$

Figure 5: Semantics (excerpt).

P, written $a \models P$, when $a \in P$. For two sets A and B, a relation R is an $A \times B$ predicate. We use infix notations for relations. State predicates are ($H^\sharp \times H \times Lenv \times LinState$) predicates, specifying shared memories and intra-thread states. A shared memory interference is a binary relation on $H^\sharp \times H$, and is used for relies and guarantees. We refer to both state predicates and shared memory interferences as *assertions*.

The rely-guarantee reasoning is done at the intermediate level $\mathcal{L}^\flat$, on instrumented programs, more precisely on the hybrid code of abstract methods implementations. Hence, assertions specify properties about the concrete and abstract heaps simultaneously.

Our work derives a compiler correctness result from a *generic* rely-guarantee specification. Of course, this cannot be achieved for an arbitrary RG specification, so we need to constrain this specification. In the following, we explain exactly what this specification looks like, and means.

4.1 Semantic RG Judgment

A hybrid method m must be specified with a semantic RG judgment of the form $R, G, I \models_{\mathtt{m}} \{P\}\ c\ \{Q\}$, where P, Q, I are state predicates, R and G are shared memory interferences, and c is the body of method m. State predicate I is meant to specify the coherence invariant between abstract objects and their representation in the concrete heap. It is asked to be proved invariant separately (see Definition 4.4). The RG judgment intuitively states that starting in a state satisfying P and invariant I, interleaving c with an environment behaving as prescribed by R (written $\rightarrow_R^*$), leads to a state satisfying Q. Additionally, this execution of c must be fully reflected by guarantee G. This intuition, typical of RG reasoning, is formalized by the first two items below.

Definition 4.1. *Judgment $R, G, I \models_{\mathtt{m}} \{P\}\ c\ \{Q\}$ holds whenever:*

1. *The post-condition is established from pre-condition and invariant:*
   $(\langle \mathtt{m}, \mathtt{c}, l, ls\rangle, \sigma) \rightarrow_R^* (\langle \mathtt{m}, \bullet, l', ls'\rangle, \sigma')$
   and $(l, ls, \sigma) \models P \cap I$
   implies $(l', ls', \sigma') \models Q$
2. *Instructions comply with the guarantee:*
   $(\langle \mathtt{m}, \mathtt{c}, l, ls\rangle, \sigma) \rightarrow_R^* (\langle \mathtt{m}, \mathtt{c}', l', ls'\rangle, \sigma') \rightarrow (\langle \mathtt{m}, \mathtt{c}'', l'', ls''\rangle, \sigma'')$
   and $(l, ls, \sigma) \models P \cap I$
   implies $\sigma'\ G\ \sigma''$
3. *Linearization points are unique and non-blocking:*
   $(\langle \mathtt{m}, \mathtt{c}, l, ls\rangle, \sigma) \rightarrow_R^* (\langle \mathtt{m}, \mathtt{Lin(b)}, l', ls'\rangle, (h^\sharp, h))$
   and $(l, ls, \sigma) \models P \cap I$
   and $[\![\mathtt{b}]\!]l' = true$
   implies their exist r, a, a', v, v',
   such that $h^\sharp(r) = a$
   and $[\![\mathtt{m}]\!]^\sharp(a, v) = (a', v')$
   and $ls' = \mathsf{Before}(r, v)$

The last condition above is novel, and more subtle than the others. It captures a necessary requirement to ensure that Lin instructions do not block programs ($[\![\mathtt{m}]\!]^\sharp$ is defined), and are unique (the linearization state is Before). This condition is essential to ensure that we can clean up the Lin instrumentation of hybrid programs, and that our semantic refinement is not vacuously true.

4.2 Specifying Hybrid Methods

We now explain the specific RG judgment we require for hybrid methods.

The above RG judgment involves state predicates and shared memory interferences. In fact, we build them from elementary bricks, *object predicates* and *object interferences*, that consider one object — one instance of a data structure — at a time, pointed to by a given reference $r \in Ref$.

DEFINITION 4.2. *An* object predicate P_r *is a predicate on pairs of an abstract object and a concrete heap:* $P_r \subseteq A^\sharp \times H$. *An* object interference R_r *is a relation on pairs of an abstract object and a concrete heap:* $R_r \subseteq (A^\sharp \times H) \times (A^\sharp \times H)$.

EXAMPLE 4.1. *The coherence invariant specifies that an abstract* `Locked` *(resp.* `Unlocked`*) lock is implemented in the concrete heap as an object whose field* `flag` *is set to* `1` *(resp.* `0`*). It is formalized as the following object predicate:*

$$\begin{aligned} ILock_r \triangleq\ & \{(\texttt{Locked}, h) \mid h(r, \texttt{flag}) = 1\} \\ & \cup \{(\texttt{Unlocked}, h) \mid h(r, \texttt{flag}) = 0\} \end{aligned}$$

Object guarantees for `acquire` *and* `release` *express the effect of the methods on the shared memory when called on a reference* r*. They are defined as the following object interferences.*

$$G^r_{\texttt{rel}} \triangleq \{((a, h_1), (\texttt{Unlocked}, h_2)) \mid h_2 = h_1[r, \texttt{flag} \leftarrow 0]\}$$

$$\begin{aligned} G^r_{\texttt{acq}} \triangleq & \{((\texttt{Unlocked}, h_1), (\texttt{Locked}, h_2)) \mid \\ & h_1(r, \texttt{flag}) = 0 \text{ and } h_2 = h_1[r, \texttt{flag} \leftarrow 1]\} \end{aligned}$$

In $G^r_{\texttt{acq}}$*, the assignment to* `flag` *is performed only if the* `cas` *succeeds.*
Finally, both `acquire` *and* `release` *have the same object rely, when called on a reference* r*: indeed, another thread could call both methods on the same reference. So we define the following object interference:* $R^r_{\texttt{m}} \triangleq G^r_{\texttt{rel}} \cup G^r_{\texttt{acq}}$*, for* $\texttt{m} \in \{\texttt{rel}, \texttt{acq}\}$.

We now need to lift object predicates and object interferences to state predicates and shared-memory interferences to enunciate the RG specifications of hybrid implementations. The challenge here is twofold: make the specification effort relatively light for the user, and, more importantly, sufficiently control the specifications so that we can derive our generic result.

An object predicate P_r is lifted to a state predicate $\widehat{P_r}$ by further specifying that, in the abstract heap, r points to an abstract object satisfying P_r:

$$\widehat{P_r} = \{(h^\sharp, h, l, ls) \mid \exists a, h^\sharp(r) = a \text{ and } (a, h) \models P_r\}$$

Similarly, in order to lift an object guarantee G_r to a shared memory interference $\widehat{G_r}$, the reference r should point to an abstract object in the abstract heap. Moreover, its effect on this object should be reflected in the resulting abstract heap. Formally:

$$\begin{aligned} \widehat{G_r} = \ & \{((h^\sharp, h_1), (h^\sharp[r \mapsto a_2], h_2)) \mid \\ & \exists a_1, h^\sharp(r) = a_1 \text{ and } (a_1, h_1)\ G_r\ (a_2, h_2)\} \end{aligned}$$

Lifting relies is a bit more subtle. When executing an hybrid implementation `m`, one should account for two kinds of concurrent effects: the client code, and the rely of the method itself. To model the client code effect, we introduce a public shared memory interference, written R_{pub}, that models any possible effect on the concrete heap, except modifying private fields in $\mathcal{P}$:

$$\begin{aligned} R_{pub} = \ & \{((h^\sharp, h_1), (h^\sharp, h_2)) \mid \\ & \forall r, f,\ f \in \mathcal{P} \Rightarrow h_1(r, f) = h_2(r, f)\} \end{aligned}$$

As for the method's rely R_r, we should consider that it could occur on *any* abstract object present in the abstract heap. Hence, a lifted rely $\widetilde{R}$ includes (i) the client public interference, and (ii) the method's rely R_r quantified over all r:

$$\begin{aligned} \widetilde{R} = R_{pub} \cup \{((h^\sharp_1, h_1), (h^\sharp_2, h_2)) \mid \exists r, a_1, a_2, \\ h^\sharp_1(r) = a_1 \\ \text{and } h^\sharp_2 = h^\sharp_1[r \mapsto a_2] \\ \text{and } (a_1, h_1)\ R_r\ (a_2, h_2)\} \end{aligned}$$

Before we define the RG proof obligation asked of hybrid method implementations, let us first recall the usual RG definition of stability.

DEFINITION 4.3. *State predicate* P *is* stable *w.r.t. shared memory interference* R *if* $\forall l, ls, \sigma_1, \sigma_2,,$

$$(\sigma_1, l, ls) \models P \text{ and } (\sigma_1 R\ \sigma_2) \text{ implies } (\sigma_2, l, ls) \models P$$

Now, we fix an invariant I_r. For a method $\texttt{m} \in \mathcal{I}$, let $G^r_{\texttt{m}}$ and $R^r_{\texttt{m}}$ be the object guarantee and rely of `m`, as previously illustrated in Example 4.1. An RG specification for `m` includes an RG semantic judgment, and stability obligations:

DEFINITION 4.4 (RG METHOD SPECIFICATION). *The RG specification for method* $\texttt{m} \in \mathcal{I}$ *includes the three following conditions:*

- *For all* $r \in Ref$*,* $\widetilde{R}_{\texttt{m}}, \widehat{G^r_{\texttt{m}}}, \widehat{I_r} \models_{\texttt{m}} \{\mathbb{P}_r\}\ \texttt{m}.comm\ \{\mathbb{Q}_r\}$
- *For all* $r \in Ref$*, predicate* $\widehat{I_r}$ *is stable w.r.t.* $\widetilde{R}_{\texttt{m}}$
- *For all* $r, r' \in Ref$*, predicate* $\widehat{I_r}$ *is stable w.r.t.* $\widehat{G^{r'}_{\texttt{m}}}$

In the above judgment, we impose the pre- and post-condition $\mathbb{P}_r$ and $\mathbb{Q}_r$. $\mathbb{P}_r$ expresses that (i) r points to an abstract object in the abstract heap, (ii) the linearization state is set to Before and (iii) the reserved local variable `this` of method `m` is set to r. $\mathbb{Q}_r$ expresses that (i) the linearization state is set to After, and (ii) the value virtually returned by the abstract method (when encountering the `Lin` instruction) matches the value returned by the concrete code. Intuitively, stability requirements ensure that $\widehat{I_r}$ is indeed an invariant of the whole program.

4.3 Compiler Correctness Theorem

So far, we have expressed requirements on hybrid methods, each taken in isolation. The last requirement we formulate is the consistency between relies and guarantees of methods. For a method `m`, to ensure that $R_{\texttt{m}}$ is indeed a correct over-approximation of its environment, we ask that $R_{\texttt{m}}$ includes any guarantee $G_{\texttt{m}'}$, where $\texttt{m}'$ is a method that may be called concurrently to `m`. This requirement is formalized by the following definition.

DEFINITION 4.5 (RG CONSISTENCY). *For all threads* t, t' *such that* $t \neq t'$*, all methods* $\texttt{m}, \texttt{m}' \in \mathcal{I}$ *and all* $r \in Ref$*,* $\mathsf{is\_called}(t, \texttt{m}) \wedge \mathsf{is\_called}(t', \texttt{m}') \Rightarrow G^r_{\texttt{m}'} \subseteq R^r_{\texttt{m}}$ *where* $\mathsf{is\_called}(t, \texttt{m})$ *indicates that* `m` *appears syntactically in the code of* t.

Relying on predicate is_called allows for accounting for data structures used according to an elementary protocol (such as single-pusher, single-reader buffers).

We finally package the formal requirements on hybrid implementations into the RGspec judgment and use it to state our main result, establishing that the target program semantically refines the source program.

DEFINITION 4.6. *Let $\mathcal{I} = \{\mathtt{m}_1 \ldots, \mathtt{m}_n\}$. $\mathcal{I}$ satisfies* RGspec*, written* RGspec($\mathcal{I}$)*, if $\forall\, i \in [1, n]$, an RG method specification is provided for $\mathtt{m}_i$, and RG consistency holds.*

THEOREM 4.1 (COMPILER CORRECTNESS). *Let σ_i an initial shared memory satisfying the invariant I_r for all $r \in Ref$ allocated in it. If* RGspec($\mathcal{I}$)*, then*

$$\forall p \in \mathcal{L}^\sharp, \mathtt{obs}(\mathtt{compile}\langle\mathcal{I}\rangle(p), \sigma_i) \subseteq \mathtt{obs}(p, \sigma_i)$$

We emphasize that the client program p is arbitrary, modulo some basic syntactical well-formedness conditions (e.g., no private field is accessed in the client code, or methods in $\mathcal{I}$ do not modify public fields).

Theorem 4.1 is phrased and proved *w.r.t.* an RG semantic judgment. In our formal development, we have developed a sound, syntax-directed proof system to discharge the RG semantic judgment, and have successfully used this system to prove the implementation of the spinlock, as well as the concurrent buffer data structure.

5. PROOF OF THE MAIN THEOREM

Theorem 4.1 is proved by establishing, from the RGspec($\mathcal{I}$) hypothesis, a simulation between the source program and its compilation. Here, we use the terminology of *backward* simulation, as is standard in the compiler verification community [21].

DEFINITION 5.1. *A relation $\backsim$ between states of $\mathcal{L}$ and $\mathcal{L}^\sharp$ is a* backward simulation *from $\mathcal{L}^\sharp$ to $\mathcal{L}$ when, for any s_1, s_2 and $s_1^\sharp$, if $s_1^\sharp \backsim s_1$ and $s_1 \xrightarrow{o} s_2$, then there exists $s_2^\sharp$ such that $s_1^\sharp \xrightarrow{o}{}^* s_2^\sharp$, and $s_2^\sharp \backsim s_2$.*

Eliding customary constraints on initial and final states, such a simulation entails preservation of observable behaviors (Theorem 3 in [21]). We establish two backward simulations, from $\mathcal{L}$ to $\mathcal{L}^\flat$ and from $\mathcal{L}^\flat$ to $\mathcal{L}^\sharp$, which we compose.

5.1 Leveraging RGspec($\mathcal{I}$)

The key point is to carry, within the simulation, enough information to leverage RGspec($\mathcal{I}$). This is necessary for both simulations, so we factorize the work by expressing a rich semantic invariant $\mathbb{I}$ over the execution of the $\mathcal{L}^\flat$ program. To simplify its definition and its proof, $\mathbb{I}$ is built as a combination of thread-local invariants.

For a thread t, the invariant $\mathbb{I}_t$ includes three kinds of information. First, it ensures various well-formedness properties of intra-thread states. Second, it demands that $\widehat{I}_r$, the coherence invariant, holds for all r. The third information is more subtle. When executing a hybrid method $\mathtt{m}$ called on a reference r, to leverage its specification $\widetilde{R}_\mathtt{m}, \widehat{G_\mathtt{m}^r}, \widehat{I}_r \models_\mathtt{m} \{\mathbb{P}_r\}\,\mathtt{m}.comm\,\{\mathbb{Q}_r\}$, we keep track, in $\mathbb{I}_t$, that the state is reachable from a state satisfying $\widehat{I}_r$ and $\mathbb{P}_r$. Recall that Definition 4.1 uses the abstract semantics $\rightarrow_{\widetilde{R}_\mathtt{m}}$. When defining $\mathbb{I}_t$, we generalize $\rightarrow_{\widetilde{R}_\mathtt{m}}$ into relation $\rightarrow_{R_t}$, where $R_t \triangleq \bigcap_{\mathtt{m} \in \mathsf{is\_called}(t)} \widetilde{R}_\mathtt{m}$. Rely R_t overapproximates interferences of threads concurrent to t, while being precise enough to deal with any method $\mathtt{m}$ called by t, since $R_t \subseteq \widetilde{R}_\mathtt{m}$.

We define $\mathbb{I} \triangleq \{(\gamma, \sigma) \mid \forall t, (\gamma(t), \sigma) \models \mathbb{I}_t\}$. To prove that $\mathbb{I}$ holds on the interleaving semantics, we first prove that all the $\mathbb{I}_t$ are preserved by the intra-thread steps. Besides, we prove their preservation by other threads' steps:

LEMMA 5.1. *Let γ, σ and $t \neq t'$ be such that $(\gamma(t), \sigma) \models \mathbb{I}_t$ and $(\gamma(t'), \sigma) \models \mathbb{I}_{t'}$. If $(\gamma(t'), \sigma) \xrightarrow{o} (ts', \sigma')$, then $(\gamma(t), \sigma') \models \mathbb{I}_t$.*

5.2 Simulation Relations

For both compilation phases, we build an intra-thread, or *local*, simulation that we then lift at the inter-thread level. Both relations are defined using the same pattern: in a pair of related states, the $\mathcal{L}^\flat$ state satisfies $\mathbb{I}_t$. It remains to encode in the relation the matching between execution states.

For the first compilation phase, a whole execution of an hybrid method is simulated by a single abstract step, occurring at the linearization point. We therefore build a *1-to-0/1* backward simulation.

Relation $\backsim_\flat^\sharp$ states that shared memories are equal on the heaps domains in $\mathcal{L}^\sharp$. Local environments are trickier to relate. In client code, they simply are equal. During a hybrid method call $\mathtt{x} = \mathtt{y.m(z)}$, before the `Lin` point, the abstract environment is equal to the environment of the $\mathcal{L}^\flat$ caller. After the `Lin` point, the only mismatch is on variable $\mathtt{x}$, which has been updated in $\mathcal{L}^\sharp$, but not yet in $\mathcal{L}^\flat$.

Proving that $\backsim_\flat^\sharp$ is a simulation follows the above three phases. Steps by client code are matched *1-to-1*; inside a hybrid method, steps match *1-to-0* until the `Lin` point; the `Lin` step is matched *1-to-1*; after the `Lin` point, steps match *1-to-0* until the return instruction. At method call return, we use RGspec($\mathcal{I}$) via $\mathbb{I}_t$, and in particular $\mathbb{Q}_r$, to prove that environments coincide on $\mathtt{x}$ again.

When simulating from $\mathcal{L}$ to $\mathcal{L}^\flat$, `Lin` instructions have been replaced by a $\bullet$. It is therefore a *1-to-1* backward simulation. Recall that $\mathcal{L}$ semantics contains no *LinState* nor abstract heap. Relation $\backsim^\flat$ therefore only states the equality of local environments and concrete heaps.

Proving this simulation is what makes the third item in Definition 4.1 necessary. Indeed, we can match the $\bullet$ step in the $\mathcal{L}$ program only if the `Lin` instruction in the $\mathcal{L}^\flat$ program is non-blocking.

The independent proof of the invariant simplifies the lifting of simulations. Indeed, except for the part about $\mathbb{I}_t$, that is already proved invariant by other threads' steps, relations keep track of the same information for all threads. Hence, their preservation by the interleaving essentially comes for free.

6. CASE STUDY: CONCURRENT BUFFERS

This case study is taken from our verification project of a concurrent mark-and-sweep garbage collector [31]. Describ-

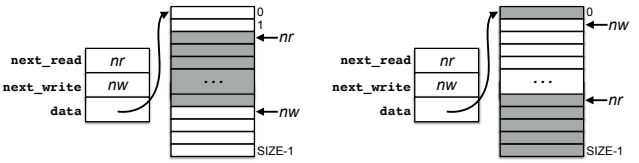

Figure 6: Concrete buffers layout (examples). Elements contained in the buffer are colored in grey. Example on the right shows how the array is populated circularly.

ing the full algorithm is out of the scope of this paper. Here, we only give an idea of why and how buffers are used.

In this algorithm, application threads, a.k.a *mutators*, are never blocked by the collector thread. They must therefore participate to the marking of potentially live objects. Buffers, so-called *mark buffers* in Domani et al.'s Java implementation [3], keep track of references to objects that are the roots of the graph of objects that may be live. Each thread, including the collector, owns a buffer. Only the owner of the buffer can push on it, and only the collector can read and pop from buffers.

In this section, we present abstract buffers and their fine-grained implementation. We also briefly explain how we express their correspondence in a coherence invariant, as well as methods guarantees and relies, and how we encode their usage protocol, which is crucial to their correctness. Buffers are fully verified in our formal development [30]: their implementation satisfies the RGspec predicate, and by Theorem 4.1, any program using abstract buffers can be semantically refined into a program in $\mathcal{L}$.

6.1 Abstract Buffers

An abstract buffer is a queue of bounded size *SIZE*, that we model by a list (of type `list value`). Below, we use the usual lists constructors. A buffer is pushed on one end of the list, and popped off from its other end. Buffers provide four methods: `isEmpty`, `top`, `pop`, and `push`. Their respective abstract semantics are[3]:

$$\begin{aligned}
[\![\texttt{isEmpty}]\!]^\sharp(ab, v) &= (ab, 1) \text{ if } ab = nil \\
&= (ab, 0) \text{ otherwise} \\
[\![\texttt{top}]\!]^\sharp(x :: ab, v) &= (x :: ab, x) \\
[\![\texttt{pop}]\!]^\sharp(x :: b, v) &= (b, Null) \\
[\![\texttt{push}]\!]^\sharp(b, v) &= (b{+}{+}[v], Null) \text{ if } |b| < SIZE - 1.
\end{aligned}$$

6.2 Fine-grained Implementation

The fine-grained implementation we prove is similar to that of Domani et al. [3], except that we use bounded-sized buffers. Buffers are objects with three fields (see Figure 6). Field `data` contains a reference to an array of fixed size *SIZE*, containing the elements of the buffer. Two other fields, `next_read` and `next_write`, indicate the bounds, within the array, of the effective content of the buffer. Field `next_read` contains the array index from which to read, while `next_write` contains the index of the first free slot in the array.

[3]The input argument v is relevant only for method `push`.

```
def isEmpty() ::=
  nr=this.next_read;
  atomic ⟨ nr=this.next_write; Lin(true) ⟩;
  return (nw==nr)

def top() ::=
  nr=this.next_read;
  nr=this.next_write;
  assume(nr<>nw); // buffer not empty
  d=this.data;
  res=d[nr];
  atomic ⟨ Lin(true) ⟩;
  return res

def pop() ::=
  nr=this.next_read;
  nw=this.next_write;
  assume (nr<>nw); // buffer not empty
  nr=(nr+1) mod SIZE;
  atomic ⟨ this.next_read=nr; Lin(true) ⟩;
  return

def push(v) ::=
  nw=this.next_write;
  nr=this.next_read;
  d=this.data;
  d[nw]=v;
  nw=(nw+1) mod SIZE;
  assume (nr<>nw); // no buffer overflow
  atomic ⟨ this.next_write=nw; Lin(true) ⟩;
  return
```

Figure 7: Buffers in $\mathcal{L}^\flat$.

A buffer is empty if and only if `next_read` and `next_write` are equal. Pushing a value on a buffer consists in writing this value in the array, at position `next_write`, and then incrementing `next_write`. Conversely, popping a value from a buffer is done by incrementing `next_read`. To consult its top value, one reads the array element at position `next_read`. In fact, the `data` array can be populated in a modulo fashion (see right example in Figure 6). The code for implementing buffers is given in Figure 7, and follows the above principles. Our core language does not include proper arrays, but we encode them with appropriate macros. The code is blocking when trying to pop on an empty buffer, or trying to push on a full buffer, as is the case for the abstract version. This is no limitation in practice: the size of buffers is chosen at initialization time, and can be upgraded at will.

6.3 Invariant, Guarantees and Relies

The coherence invariant between an abstract buffer and its concrete implementation essentially consists in that every cell in the `data` array between `next_read` and `next_write` matches, in order, the elements of the list representing the abstract buffer – in particular, the number of valid cells in the array must be equal to the length of the abstract buffer.

We express it as the following object predicate, where function $range(s, e)$ computes the list of successive indices be-

tween two integers s and e, modulo $SIZE$.

$$
\begin{aligned}
&IBuck_r \subseteq (\texttt{list value}) \times H\\
&IBuck_r \triangleq \{(b,h) \mid b = b_1 :: \ldots :: b_s \quad s{<}SIZE\\
&\qquad\qquad Separation(h, \texttt{data})\\
&\qquad\qquad range(nr, nw) = i_1 :: \ldots :: i_s\\
&\qquad\qquad h(r, \texttt{next\_write}) = nw \quad nw{<}SIZE\\
&\qquad\qquad h(r, \texttt{next\_read}) = nr \quad nr{<}SIZE\\
&\qquad\qquad \forall j \in \{1, ..s\},\ h(h(r, \texttt{data}), i_j) = b_j \qquad \}
\end{aligned}
$$

The property $Separation(h, \texttt{f})$ states that all values of `f` fields in concrete heap h are unique. In the definition of $IBuck_r$, this encodes the separation of arrays (all `data` field are distinct references), thus reflecting the ownership of buffers by each thread.

Methods guarantees reflect each basic shared memory effect of the methods (see [30] for details). Methods `top` and `isEmpty` have no effect: their object guarantee $G^r_{\texttt{top}}$ and $G^r_{\texttt{isEmpty}}$ is the identity. Methods `pop` and `push` have been explained informally previously. Their guarantee $G^r_{\texttt{pop}}$ and $G^r_{\texttt{push}}$ also reflect the effect on the abstract buffer, and this effect is specified to occur atomically with the instruction annotated by the `Lin` instruction (see Figure 7).

Finally, we express the usage protocol of single-writer, single-reader by defining the respective rely of methods.

$$R^r_{\texttt{isEmpty}} \triangleq G^r_{\texttt{push}} \quad R^r_{\texttt{pop}} \triangleq G^r_{\texttt{push}} \quad R^r_{\texttt{top}} \triangleq G^r_{\texttt{push}} \quad R^r_{\texttt{push}} \triangleq G^r_{\texttt{pop}}$$

With the above definitions, and using the general approach we have illustrated on the lock in Section 4, we were able to formally establish $\mathsf{RGspec}(\{\texttt{isEmpty}, \texttt{top}, \texttt{pop}, \texttt{push}\})$, and then apply our generic refinement theorem, establishing the correctness of the compiler specialized to buffers. We refer the reader to our formal development [30] for the full details of the proof.

7. RELATED WORK

The literature on linearizability verification is vast. Dongol et al. [4] provide a comprehensive survey of techniques for verifying linearizability *w.r.t.* the seminal definition of Herlihy [16]. Notably, a number of works use concurrent program logics, most of them influenced by Jones's rely-guarantee [18] and O'Hearn's Concurrent Separation Logic [24]. Both ideas have been combined into logics like RGSep [29], SAGL [5], and more recently Iris [19]. Another logic is worth mentioning, although not directly applied to linearizability: Sergey et al. [26] provide a Coq framework to mechanically verify fine-grained concurrent algorithms, based on the FCSL logic. FCSL's soundness is formally proved in Coq *w.r.t.* a denotational semantics, but the shallow embedding of programs in Coq makes the approach hard to use in compiler verification. While the above logics have highly expressive powers, they are not formally linked with observational refinement, which is what we aim at. Filipović et al. [6] characterize linearizability in terms of observable refinement, on top of a non-operational semantics. Here, we want to express our main result in terms of observable refinement directly.

Our work is inspired by the technique outlined in [28]. Here, we use a simple rely-guarantee formulation: we argue it is a good balance between the logic expressivity and its mechanization effort. Indeed, it is enough for implementing and proving the concurrent garbage collector we verified in [31]: our meta-theorem applies to the verified implementation of the marking buffers for roots publishing, removing the need for a shared shadow-stack [13, 1]. For a fine-grained implementation of the freelist, our methodology is also able to handle Treiber's stack [27]. More sophisticated data structures exist that are less subject to contention, e.g. Hendler's [14], but external linearization points are left for future work.

The technique presented in [28] lacks a mechanized soundness proof that would be suitable to a verified compiler infrastructure. Our work provides such a foundation. Liang et al. [23] tackle a problem similar to ours. They define a simulation parameterized by relies and guarantees, and compositionality rules, to reason about program transformations. In [22], they combine it with the technique in [28] to verify linearizability. Though [22] is able to deal with external linearization points, their proofs remain un-mechanized.

The work of Derrick et al. [2], formalized in the KIV tool, is also closely related. Like us, they express linearizability through thread-local proof obligations, establishing systematically inter-thread simulations. Our work differs in the nature of the proof obligations: we choose to express them in terms of rely-guarantee, so we can discharge them using program logics.

Jagannathan et al. [17] propose an atomicity refinement methodology for verified compilation. Their final theorem, mechanized in Coq, is expressed as a behavior preservation, but the proof methodology is completely different from the one presented here. They provide compositional rules to symbolically refine high-level atomic blocks into fine-grained low-level code.

8. CONCLUSION

This work embeds the notion of linearizable data structures in a compiler formally verified in Coq. As such, the present work represents a mechanized soundness foundation for Vafeiadis's technique [28], phrased in terms of semantic refinement. To achieve this result, we establish, starting from proof obligations expressed in terms of rely-guarantee reasoning, a generic backward simulation theorem. Its proof requires to express subtle coherence invariants between abstract atomic primitives and their fine-grained concurrent implementation. The development size is close to $13kloc$, an effort of one person-year.

The present work is part of a larger project aiming at verifying concurrent compilation mechanisms such as garbage collectors or dynamic allocators, and plugging them in a verified compiler infrastructure. Such artifacts are notably difficult to verify, and one needs to abstract from low-level details when conducting their formal soundness proof. In particular, the concurrent garbage collector we proved in [31] is implemented in a dedicated program intermediate representation that features constructs facilitating the programming and the proof of compiler services, such as introspection on objects and high-level management of threads roots. Notably, the collector and mutators threads share abstract concurrent buffers to keep track of potentially live objects. The work pre-

sented here allows to propagate the formal soundness of the collector down to its low-level, fine-grained implementation.

9. REFERENCES

[1] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience*, 21(12), 2009.

[2] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *TOPLAS*, 2011.

[3] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanover. Implementing an on-the-fly garbage collector for Java. In *Proc. of ISMM'00*, 2000.

[4] B. Dongol and J. Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, Sept. 2015.

[5] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. of ESOP*, 2007.

[6] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *Proc. of ESOP*, 2009.

[7] P. Gammie, A. L. Hosking, and K. Engelhardt. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In *Proc. of PLDI 2015*, 2015.

[8] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Proc. of CAV*, 1996.

[9] K. Havelund. *Mechanical verification of a garbage collector*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[10] K. Havelund and N. Shankar. A mechanized refinement proof for a garbage collector, 1997. Unpublished report, available at http://havelund.com/Publications/gc-paper.ps.

[11] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *Proc of POPL*, 2009.

[12] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proc. of CAV*, 2015.

[13] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proc. of ISMM*, 2002.

[14] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proc. of SPAA*, 2004.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[17] S. Jagannathan, V. Laporte, G. Petri, D. Pichardie, and J. Vitek. Atomicity refinement for verified compilation. *TOPLAS*, 2014.

[18] C. B. Jones. Specification and design of (parallel) programs. In *IFIP*, 1983.

[19] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proc. of POPL*, 2015.

[20] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand. Closing the gap – the formally verified optimizing compiler CompCert. In *Proc. of SSS'17*, 2017.

[21] X. Leroy. A formally verified compiler back-end. *JAR*, pages 363–446, 2009.

[22] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *Proc. of PLDI*, 2013.

[23] H. Liang, X. Feng, and M. Fu. Rely-Guarantee-based simulation for compositional verification of concurrent program transformations. *TOPLAS*, 2014.

[24] P. W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 2007.

[25] D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6(4), 1994.

[26] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proc. of PLDI*, 2015.

[27] R. K. Treiber. Systems programming: Coping with parallelism, 1986. Technical Report RJ 5118. IBM Almaden Research Center.

[28] V. Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, 2007.

[29] V. Vafeiadis and M. J. Parkinson. A marriage of Rely/Guarantee and separation logic. In *Proc. of CONCUR*, 2007.

[30] Yannick Zakowski and David Cachera and Delphine Demange and David Pichardie. Verified compilation of linearizable data structures – formal development, 2017. http://www.irisa.fr/celtique/ext/simulin/.

[31] Y. Zakowski, D. Cachera, D. Demange, G. Petri, D. Pichardie, S. Jagannathan, and J. Vitek. Verifying a Concurrent Garbage Collector using a Rely-Guarantee Methodology. In *Proc. of ITP*, 2017.